STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Number 1, 2014

SYN!BAD: A SYNONYM-BASED REGULAR EXPRESSION EXTENSION FOR KNOWLEDGE EXTRACTION TASKS

OVIDIU ŞERBAN

Abstract. This paper focuses on presenting Syn!bad, a synonym-based regular expression language which can be used for basic Knowledge Extraction tasks, such as the Natural Language Understanding components for Dialogue Management. The language offers a simple syntax for complex matching processes, which brings a new solution for the input variability problem, often encountered when dealing with natural language. The language provides various special tokens to match synonym-based expressions free context variables and generic Part of Speech tokens.

1. Introduction

In the field of Natural Language Processing (NLP), manipulating data patterns has been a very successful approach. These patterns are the key to "understanding" the natural language, by grouping similar elements by their functionality. The evolution of automata theory leads to a new language to represent these patterns: the Regular Expressions [16]. There are many standards for Regular Expressions, leading to various implementations [7]. We will focus our discussion on the basic (BRE) and extended (ERE) features of Regular Expressions, because of their frequent usage.

From the task oriented algorithms perspective, regular expressions are very popular in NLP systems, either used alone or in preprocessing tasks for mixed approaches. Most of the sentence tokenization algorithms are based on regular expressions [8, 2]. Moreover, more complex tasks, such as Part-of-Speech Tagging (POS) [13] or Named Entity Recognition (NER) [10] use them as well. Recently, regular expressions are employed to process emoticons [11] for Sentiment Analysis Applications. In general, regular expressions are used in applications where robust but simple algorithms need to be employed

Received by the editors: December 1, 2013.

2010 *Mathematics Subject Classification.* 68T35, 68T50.

1998 *CR Categories and Descriptors.* H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Natural Language*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation – *Automata.*

Key words and phrases. Knowledge Extraction, Dialogue Systems, Regular Expressions.

as part of a preprocessing strategy [5]. From the application perspective, the regular expressions have been used successfully in spam detection and filtering [3] or more generic data mining techniques [17].

Syn!bad is an extended regular expression language, for usage mainly in Natural Language Processing (NLP) applications. It uses the extended POSIX Regular Expression [1] structures among others, more specific to NLP domain. Sometimes, the learning phase for the detection algorithms requires a feature extraction method. The knowledge extraction methods, involved in the Natural Language Understanding Process, use similar techniques to detect key concepts to be used in the Dialogue Management process. We propose this language to simplify the construction of these patterns.

The name, Syn!bad (also written: Synnbad, with double nn, instead of n!) is an acronym of *Synonyms [are] not bad*. This suggests that the main concepts of Syn!bad are centred among synonyms processing, using different dictionaries.

Synonyms are independent structures, grouped in different sets, by their meaning. The most common grouping currently known is the WordNet[1] synsets [9], which consists in grouping different words according to their semantics and part of speech. The basic unit, a synset (a set of synonymous words which refer to a common semantic concept), has a unique id, which permits an easy retrieval.

Syn!bad is available both as an independent library and as a component of the AgentSlang platform [14]. Nevertheless, Syn!bad is intended to be a platform and language independent library that can be implemented and distributed on its own. We present the language in the scope of basic knowledge extraction for Interactive Systems (IS), but this library can be extended to document classification, summarisation, topic extraction, etc.

In dialogue management, knowledge extraction or affect detection, building a set of patterns to extract the information simplifies the complexity of any system. Moreover, it gives a tool set flexible enough to process any data. Appendix A provides a formal view over the language, by presenting the BNF Grammar definition of Syn!bad .

This article is organized as following: we start with a brief presentation of our context, related to the Interactive System domain. We continue with the description of the system, which includes several implementation details. At last, we conclude with a discussion about the current paper.

[1]WordNet is a commonly known lexical database for English language.

2. Context of the problem

In the field of Interactive Systems (IS), the knowledge extraction process is usually slowed down by the complexity of the rules describing a certain concept. Using regular expressions is an alternative, but in certain situations, composing rules for all the cases is impossible. Another approach is to group certain structures while making them more generic. For instance, instead of using a regular expression for matching the following sentence: `Bob can I have your phone`, one could use `<name> can I <verb> your <object>`. By using regular expressions, the variable structures are already supported by certain implementations.

When adding restrictions to the matched variables the problem becomes more difficult, especially in the case of `<verb>` and `<object>`. To our knowledge, the syntax of matching only variable structures while having a certain part of speech is not supported by any regular expression implementation.

A more complex situation is given by placing a synonymic relation restriction on the matched item. In our previous example, we would like to extract only the objects being synonyms of the word *phone*. The synonyms usually introduce a certain fuzziness into a decision, since not all the meanings of a polysemantic word match the context of a given pattern. In this scenario, a certain restriction can be modelled, by adding a part of speech restriction on the word. For example, the word *phone* has multiple meanings, such as *the action of calling someone*, when employed as a verb or *telephone (object)*, when used as a noun. When matching a synonym of *telephone* with our rule, we can restrict this to only *nouns*, in which case the words *call* or *ring*, as verbs, are not matched.

Another situation is dealing with variable concepts, restricted or not by a certain format. Given the following phrase: `1000$ for a phone ???`, we observe that the amount and the target object may be *variable* but also a very important parameter involved in the decision process. Nevertheless, several restrictions can be added for the amount and the object. For example, the amount clearly needs to be a number, which can be an annotation done previous to the decision and the object can be either a generic object or a synonym of the word "phone". In the end, for all our examples, the style and format of the pattern are decided by a human coder.

3. Preliminaries

The ERE, as an extension of BRE, introduces several basic operators:

- The simple terminal matching operator, which can be a character or a sequence of characters: `aabbac`, `[a-z]bce[0-9]`. In these examples,

the matching process is successful if the sample matches either the exact sequence `aabbac` or it starts with any low-case character, continues with the `bce` sequence and ends with a digit, for the case of the next sequence.

- The alternative operator (`|` which is synonym with the **or** keyword): `seq1|seq2`. The matching process is successful if any of the sequences 1 or 2 is matched.
- The zero or one time operator (`?`), which describes an optional token.
- The zero or many times operator (`*`), which allows the token multiplication during the matching process.

Several other operators are either specific to character sequence matching (`^` - for the negation of character interval) or are equivalent to a combination of multiple operators: `seq+` $\Longleftrightarrow$ `seqseq*`.

In our previous examples, we used the synonyms of the word *phone*. In fact, a very popular structure for describing the synonym grouping is given by *synsets* (Synonym Sets), as the core part of WordNet [9]. These synsets consist in a group of synonym words, which have been annotated according to their Part of Speech (POS) and have a gloss attached to describe their meaning. For an ease of use in Computer Science applications, these synsets have a unique alphanumeric index attached. The following example is the synset of the word *phone*:

*04401088: (n) telephone, **phone**, telephone set (electronic equipment that converts sound into electrical signals that can be transmitted over distances and then converts received signals back into sounds)*

where *04401088* is the synset identification number, *(n)* states that this synset contains nouns, followed by the synonym list and the last part is the gloss. Usually, the synset id is represented as a combination between the part of speech and the identification number, becoming in this case `n#04401088` or simply `04401088n`.

Another important concept used in our work is the Part of Speech (POS) tag, which corresponds to the lexical class of each word. The process of annotating a word with its specific tag is usually called Part-of-Speech Tagging and it cannot be done on an independent word, but on a whole phrase or context. Various annotation models exist, each depending on one or multiple POS Tag sets, which are usually language dependent, since not all the languages have the same lexical classes. For English, one of the most popular Tag Set available is the Penn Part-Of-Speech Tag System [12]. For French one of the most common used tag set is the TreeTagger Part-of-Speech Tags

[15], while for Central and East European languages (including Romanian) the MULTEXT-East [6] is frequently used. For the purpose of this article, we will focus our examples on the Penn POS Tag Set, with the observation that our proposition is intended to be language independent.

We continue this paper with a simple practical example of Syn!bad syntax, which will allow us to introduce all the concepts of the proposed language.

4. The Syn!bad Extension

4.1. **A practical example.** Given the previous context, we propose a first example of a Syn!bad pattern.

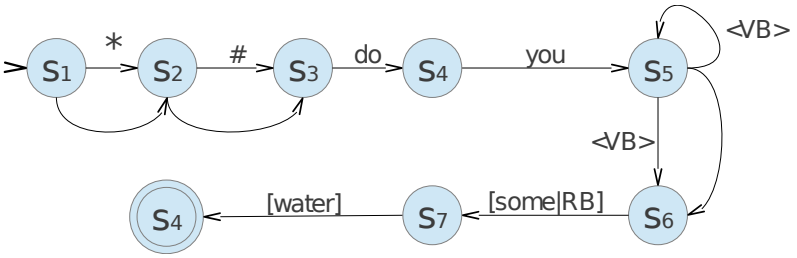

Figure 1. A Syn!bad example, presented as an automaton

Based on the rules described above, we compile the following Syn!bad pattern, which also contains most of the features of the language:

```
$name <#*>? do you <VB*>*  [some|RB*] [water#object]
```

- `$name` item represents a context free variable, which matches any single word and retrieve it as the *name* variable.
- `<#*>?` is an optional token that can match any punctuation mark. Moreover, the *#** represents a generic part of speech group matching punctuation marks.
- *do* and *you* are precise words matched by this expression.
- `<VB*>*` is a none-or-many token matcher, which restricts the element to match only a selected part of speech, in this case a verb.
- `[some|RB*]` represents a matcher for a synonym of the word *some*. Moreover, a restriction over the part of speech is added, which matches only adverbs.
- `[water#object]` this token is similar to the previous one, but a synonym of the word *water* is matched and the word is stored into the *object* variable.

In order to describe the whole matching process, the following sentence is given: `Ovidiu , do you want any aqua`

The result of the matching is: $\$name \leftarrow Ovidiu$ and $\#object \leftarrow aqua$, while `<#*>?` matches the comma mark (,), `<VB*>*` matches the single verb *want* and any is matched by the token `[any|RB*]`.

4.2. **Implementation Details.** The Syn!bad language has two levels, one is related to the grammar model, presented in the Appendix A. The second level concerns the implementation of this language, as an extension to the current capabilities of our knowledge extraction platform.

The patterns are compiled into a Deterministic Finite Automaton (DFA), completely written from scratch in Java language. We choose this representation since the DFA offers superior matching speed for a linear decision. Cox [4] presents a series of experimental arguments to sustain our choice.

The Deterministic Finite Automaton (DFA) is a type of automaton, where each state, for a given input, has at most one new state leading from the previous one. This makes the navigation through the states easier, since the possibility of exploration is always reduced to only one state or none. The finite status is given when our machine reaches one of the terminal states and the finish condition is fulfilled.

The simple token matchers use a simple word equality operator, whereas the others require more complex operators, such as part of speech matchers and synonym intersection.

Concerning the part of speech restrictions, we propose two different types of labels. One is more strict, as recommended by the Penn Part-Of-Speech Tag System [12], which contains 45 different labels. The second is a functional grouping of the first system, called Generic POS, and contains only 5 labels:

(1) `#*` groups all the punctuation marks into one single category: `$ # . , : ( ) " ’`
(2) `VB*` groups all the verb tags: `VB, VBD, VBG, VBN, VBP, VBZ`
(3) `RB*` groups all the adverb tags: `RB, RBR, RBS`
(4) `NN*` groups all the noun tags: `NN, NNS, NNP, NNPS`
(5) `JJ*` groups all the adjective tags: `JJ, JJR, JJS`

The synonyms are currently extracted from the WordNet dictionary [9]. We use the synset identifiers, provided by WordNet, restricted by Part of Speech, when necessary. WordNet provides an index already split by part-of-speech, which makes the restriction conditions much easier to fulfil.

All the part-of-speech restrictions, synonyms and variable names are stored as a matching token, making possible to model our automaton as a DFA. All the tokens of a pattern are stored as a linked multi-list.

For each state, we assign a priority to each token, which makes the matcher decision even more simple. The top priority is assigned to the *optional token*,

just before the *mandatory element*. This is done because it is more important to match an optional item, when possible, rather than a mandatory one. The process cannot continue without matching all the mandatory elements, therefore since the optional item can be skipped easily, it is important to match them before the mandatory items. The last priority is assigned to a consumer item, which is either a *skip* item or a *global variable* (a structure labelled `$name`). A *skip* is an element with the lowest priority assigned, which matches everything and it is used to define matching spaces. The current implementation uses a *skip* of 2 items defined by default.

Once the patterns are compiled, each one of them has an identifier assigned. These are not mandatory to be unique, and in certain situations it can be useful to have duplicate identifiers, such as in the case of having polysemic expressions. For instance, the patterns: (`hello`) and (`hi there`), can have the same id (`id=greeting`), since both represent different forms of greetings.

When a pattern is matched, its identifier is returned, along with all the variables matched. The variables could be global: defined as `$name`, or local: `#name` which are defined by the part-of-speech or synonym matching tokens. For instance, the pattern (`hello $name`) matches the first word that comes after `hello` and stores it in the `$name` variable, whereas the pattern `hello <NN*#name>*` matches the first *noun* that follows the word `hello` and stores it in the variable `#name`. In fact, the `$name` variable is matching any word or punctuation mark, whereas `#name` variable stores the content matched by a specific token: part-of-speech or synonym.

4.3. **Syn!bad Pattern Styles.** Patterns, among the variable retrieval feature, have another level of static labels, named styles. A style is represented by a collection of pairs (label, value) assigned to each pattern. The functional value of this feature is represented by the possibility to manually assign a second level of annotation to a certain matcher. The label space is defined on the whole matcher container (all the pattern matchers added on the same list), and the label space is sparse, as well. When a matcher does not have a label defined, an '*' is automatically assigned to any undefined value.

To introduce the *styles*, we present a short example of this functionality. Table 1 defines three different patterns, each one having different styles assigned. Styles are comma separated, defined as a label=value pair.

The pattern *p1* has two values assigned to the styles relation=familiar and rudeness=high, *p2* defines a value just for rudeness=low therefore the relation becomes `*`, *p3* has the *polite* value assigned to the relation. Table 2 summarises these results.

Pattern	**ID**	**Style**
what do you want ?	p1	relation=familiar, rudeness=high
what can i do to help you ?	p2	rudeness=low
if i may ask , how could i help you ?	p3	relation=polite

TABLE 1. Syn!bad pattern examples, using the style definition features

Style	**ID**		
	p1	p2	p3
relation	familiar	*	polite
rudeness	high	low	*

TABLE 2. The values assigned to each style according to the pattern definitions from Table 1

In another context, the styles could be used for other applications, such as sentiment detection. Given the following patters, presented in Table 3, we could observe that the pattern **p1** and **p2** are annotated with negative or positive labels, while **p3** is neutral, therefore no sentiment will be associated. During the matching process, the style for **p3** will be equal to *.

Pattern	**ID**	**Style**
This `$object` is broken !	p1	sentiment=negative
This `$object` is very good !	p2	sentiment=positive
This is a `$object` !	p3	

TABLE 3. Sentiment annotation used as Syn!bad styles

The usage of styles is not mandatory, but offers another level of granularity for the knowledge extraction model. The styles offer a complementary function for variable extraction and in case of large pattern databases, it also provides more information for the dialogue selection models and dialogue generation components.

5. Conclusion

Syn!bad is an extension to the POSIX regular expression language that employs special elements useful for NLP applications. These elements are synonymic or part-of-speech expressions that can be combined with regular word items. The patterns can be grouped into semantic clusters and have various styles assigned, which makes the matching process useful for knowledge extraction and dialogue management. In fact, this language is a critical part of the AgentSlang Platform [14], ensuring the Natural Language Understanding function of the system.

Appendix A. Syn!bad: BNF Language Specification

The BNF Grammar of the Syn!bad syntax is defined as following:

⟨*expression*⟩ ::= ⟨*token*⟩ '␣' ⟨*expression*⟩ | ⟨*token*⟩

⟨*token*⟩ ::= ⟨*pattern*⟩
| ⟨*pattern*⟩ '`*`'
| ⟨*pattern*⟩ '`?`'
| ⟨*pattern*⟩ '{' ⟨*number*⟩ '`,`' ⟨*number*⟩ '}'

⟨*pattern*⟩ ::= ⟨*word*⟩
| '`<`' ⟨*POS_Structure*⟩ '`>`'
| '`[`' ⟨*synonym*⟩ '`]`'
| '`$`' ⟨*variable*⟩

⟨*POS_Structure*⟩ ::= ⟨*POS*⟩ ('`#`' ⟨*variable*⟩)?

⟨*POS*⟩ ::= ⟨*PennPOS*⟩
| ⟨*GenericPOS*⟩

⟨*synonym*⟩ ::= ⟨*word*⟩ ('|' ⟨*POS*⟩)? ('`#`' ⟨*variable*⟩)?

⟨*number*⟩ ::= [1-9] [0-9]*

⟨*word*⟩ ::= [a-z]+

⟨*variable*⟩ ::= [a-z0-9]+

⟨*PennPOS*⟩ ::= ‘JJ’ | ‘RB’ | ‘DT’ | ‘TO’ | ‘RP’ | ‘RBR’ | ‘RBS’ | ‘LS’
| ‘JJS’ | ‘JJR’ | ‘FW’ | ‘NN’ | ‘NNPS’ | ‘VBN’ | ‘VB’ | ‘VBP’
| ‘PDT’ | ‘WP$’ | ‘PRP’ | ‘MD’ | ‘SYM’ | ‘WDT’ | ‘VBZ’ | ‘‘’
| ‘#’ | ‘WP’ | ‘’’ | ‘IN’ | ‘$’ | ‘VBG’ | ‘EX’ | ‘POS’ | ‘(’
| ‘VBD’ | ‘)’ | ‘.’ | ‘,’ | ‘UH’ | ‘NNS’ | ‘CC’ | ‘CD’ | ‘NNP’
| ‘PP$’ | ‘:’ | ‘WRB’

⟨*GenericPOS*⟩ ::= ‘#*’ | ‘VB*’ | ‘RB*’ | ‘NN*’ | ‘JJ*’

References

[1] V. Alfred. Algorithms for finding patterns in strings. *Handbook of Theoretical Computer Science: Algorithms and complexity*, pages 255 – 300, 1990.

[2] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. O'Reilly Media, 2009.

[3] Eric Conrad. Detecting spam with genetic regular expressions. *SANS Institute InfoSec Reading Room*, 2007.

[4] R. Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). `http://swtch.com/~rsc/regexp/regexp1.html`, January 2007.

[5] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Text Processing with GATE (Version 6)*. 2011.

[6] Tomaž Erjavec. Multext-east: morphosyntactic resources for central and eastern european languages. *Language resources and evaluation*, 46(1):131–142, 2012.

[7] Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7. ISO/IEC/IEEE 9945:2009, September 2009.

[8] Christopher Manning, Tim Grow, Teg Grenager, Jenny Finkel, and John Bauer. Stanford tokenizer. `http://nlp.stanford.edu/software/tokenizer.shtml`, 2010.

[9] G.A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[10] Diego Mollá, Menno Van Zaanen, and Daniel Smith. Named entity recognition for question answering. *Proceedings of ALTW*, pages 51–58, 2006.

[11] Christopher Potts. A Twitter-aware Tokenizer from the Sentiment Symposium Tutorial. `http://sentiment.christopherpotts.net/`, November 2011.

[12] B. Santorini. Part-of-speech tagging guidelines for the penn treebank project (3rd revision). Technical report, University of Pennsylvania, 1990.

[13] Helmut Schmid. Improvements in part-of-speech tagging with an application to german. In *In Proceedings of the ACL SIGDAT-Workshop*. Citeseer, 1995.

[14] Ovidiu Serban and Alexandre Pauchet. Agentslang: A fast and reliable platform for distributed interactive systems. In *Intelligent Computer Communication and Processing (ICCP), 2013 IEEE International Conference on*, pages 35–42. IEEE, 2013.

[15] Achim Stein. French TreeTagger Part-of-Speech Tags. `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/data/french-tagset.html`, April 2003.

[16] C. Kleene Stephen. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3 – 41, 1956.

[17] Daniela Xhemali, Chris J Hinde, and Roger G Stone. Genetic evolution of regular expressions for the automated extraction of course names from the web. In *GEM*, pages 118–124, 2010.

LITIS Laboratory, INSA de Rouen, Avenue de l'Université – BP 8, Saint-Étienne-du-Rouvray Cedex, 76801 France

E-mail address: `ovidiu.serban@insa-rouen.fr`